

Secure Information Flow and Pointer Confinement in a Java-like Language

Anindya Banerjee and David A. Naumann

ab@cis.ksu.edu, naumann@cs.stevens-tech.edu

Kansas State University and Stevens Institute of Technology

The Problem

- ◆ System with High and Low inputs, $L \leq H$.
 $H \equiv$ secret/private/classified
- ◆ L users permitted to see L outputs.

(Security Policy: Confidentiality \equiv “PROTECT SECRETS”)

Formalise for systems programmed in Java-like languages.

The Problem

- ◆ System with High and Low inputs, $L \leq H$.
 $H \equiv$ secret/private/classified
- ◆ L users permitted to see L outputs.

(Security Policy: Confidentiality \equiv “PROTECT SECRETS”)

Formalise for systems programmed in Java-like languages.

- ◆ Noninterference (NI) [Goguen-Meseguer '82]
“No matter how H inputs change, L outputs remain same”.
- \equiv No information flow from H to L.

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
 - ◆ private fields, class-based visibility
 - ◆ mutually recursive classes, methods
 - ◆ pointers, mutable state, dynamic allocation
 - ◆ inheritance, dynamic dispatch

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system
 - ◆ Data flow (via mutable fields)

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system
 - ◆ Data flow (via mutable fields)
 - ◆ Control flow (via dynamic dispatch)

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system
 - ◆ Data flow (via mutable fields)
 - ◆ Control flow (via dynamic dispatch)
- ◆ Proof of Noninterference (denotational semantics, compositional proofs)

What We Have Not Done

- ◆ Extension to full **JavaCard**
 - ◆ Exceptions
 - ◆ Protected fields, private/protected classes, interfaces, packages
- ◆ Extension to full **Java**
 - ◆ Threads
 - ◆ Class loading, Reflection, Native methods
 - ◆ Generics
 - ◆ ...

Previous Work: Main Inspirations

- ◆ Noninterference: Goguen-Mesequer, Denning-Denning
- ◆ Type-based analyses for information flow:
 - 1996– Smith, Volpano (Simple Imperative Language)
 - 1999– Abadi et al. (DCC – Info. flow as dependence analysis)
 - 1999– Sabelfeld, Sands (Threads, Poss. NI, Prob. NI)
 - 1999 Myers (Java – but NI open)
 - 2000– Pottier, Simonet, Conchon (Core ML)

Previous Work: Main Inspirations

- ◆ Abstract Interpretation based analyses for info. flow:
1992 Mizuno, Schmidt (**Logical relations to prove NI**)

Previous Work: Main Inspirations

- ◆ Abstract Interpretation based analyses for info. flow:
1992 Mizuno, Schmidt (**Logical relations to prove NI**)

Our focus:

- ◆ Type-based analyses for information flow: (Smith, Volpano)
 - ◆ Cope with threads, non-determinism, stochastic processes, declassification, ...
 - ◆ **Cope with realistic programming languages.**

Example: Aliasing (1)

```
class LPatient extends Object { //basic patient record
  String name;
  String getName() {return this.name;}
  unit setName(String n) {this.name := n;} }
```

Example: Aliasing (1)

```
class LPatient extends Object { //basic patient record
  String name;
  String getName() {return this.name;}
  unit setName(String n) {this.name := n;} }

class XPatient extends LPatient {
  String hiv; //SECRET
  String getHIV() {return this.hiv;}
  unit setHIV(String s) {this.hiv := s;} }
```

Example: Aliasing (1)

```
LPatient lp := readFile();  
String lBuf := lp.getName(); String hBuf := lp.getName();  
LBuf ~ lp.name ~ HBuf  
XPatient xp := new XPatient(); xp.setName(LBuf);  
LBuf ~ lp.name ~ HBuf ~ xp.name
```

Example: Aliasing (1)

```
LPatient lp := readFile();  
String lBuf := lp.getName(); String hBuf := lp.getName();  
LBuf ~ lp.name ~ HBuf  
  
XPatient xp := new XPatient(); xp.setName(LBuf);  
LBuf ~ lp.name ~ HBuf ~ xp.name  
  
String hBuf := readFromTrustedChannel(); xp.setHIV(HBuf);  
HBuf ~ xp.hiv;          LBuf ~ lp.name ~ xp.name
```

Example: Aliasing (1)

```
LPatient lp := readFile();  
String lBuf := lp.getName(); String hBuf := lp.getName();  
LBuf ~ lp.name ~ HBuf  
XPatient xp := new XPatient(); xp.setName(LBuf);  
LBuf ~ lp.name ~ HBuf ~ xp.name  
String hBuf := readFromTrustedChannel(); xp.setHIV(HBuf);  
HBuf ~ xp.hiv; LBuf ~ lp.name ~ xp.name  
LBuf := HBuf; lp.setName(xp.getHIV())
```

Example: Aliasing (1)

```
LPatient lp := readFile();
String lBuf := lp.getName(); String hBuf := lp.getName();
LBuf ~ lp.name ~ HBuf
XPatient xp := new XPatient(); xp.setName(LBuf);
LBuf ~ lp.name ~ HBuf ~ xp.name
String hBuf := readFromTrustedChannel(); xp.setHIV(HBuf);
HBuf ~ xp.hiv;          LBuf ~ lp.name ~ xp.name
LBuf := hBuf; lp.setName(xp.getHIV())
xp.name ~ xp.hiv
```

Annotated Types Prevent Direct Data Flows

```
class LPatient extends Object {  
    (String, L) name;  
    (String, L) getName() {return this.name;}  
    (unit, L) setName((String, L) n) {this.name := n;}  
class XPatient extends LPatient {  
    (String, H) hiv; //SECRET  
    (String, H) getHIV() {return this.hiv;}  
    (unit, L) setHIV((String, H) s) {this.hiv := s; }  
}
```

No direct assignment from **H** to **L**

Example: Aliasing (1) Revisited

```
(LPatient, L) lp := readFile();  
(String, L) lBuf := lp.getName();  
(String, H) hBuf := lp.getName();  
  
(XPatient, L) xp := new XPatient();   xp.setName(lBuf:L);  
(String, H) hBuf := readTrusted...;   xp.setHIV(hBuf:H);
```

Example: Aliasing (1) Revisited

```
(LPatient, L) lp := readFile();  
(String, L) lBuf := lp.getName();  
(String, H) hBuf := lp.getName();  
  
(XPatient, L) xp := new XPatient(); xp.setName(LBuf:L);  
(String, H) hBuf := readTrusted...; xp.setHIV(HBuf:H);
```

```
LBuf:(String,L) := HBuf:(String,H)
```

```
lp.setName(xp.getHIV():(String,H))
```

Example: Aliasing (1) Revisited

```
class LPatient L extends Object {
    (String, L) name;
    (String, L) getName() {return this.name;}
    (unit, L) setName((String, L) n) {this.name := n;} }

class XPatient L extends LPatient {
    (String, H) hiv; //SECRET
    (String, H) getHIV() {return this.hiv;}
    (unit, L) setHIV((String, H) s) {this.hiv := s; } }
```

Example: Aliasing (2)

```
class LPatient L extends Object {
    //name, getName, setName
    (String, L) passSelf() {
        ...o.m(this)... // m has L argument}}
class XPatient L extends LPatient { //hiv, getHIV, setHIV }
```

Example: Aliasing (2)

```
class LPatient L extends Object {
    //name, getName, setName
    (String, L) passSelf() {
        ...o.m(this)... // m has L argument}}

class XPatient L extends LPatient { //hiv, getHIV, setHIV}

class HPatient H extends XPatient { //inherits passSelf() }
```

Example: Aliasing (2)

```
class LPatient L extends Object {  
    ...  
    (String, L) passSelf() {...o.m(this)...}  
class XPatient L extends LPatient {...}  
class HPatient H extends XPatient {  
    //inherits passSelf()  
}
```

- ◆ Require: **H**-subclass of **L**-class overrides all inherited methods.

Example: Aliasing (2)

```
class LPatient L extends Object {  
    ...  
    (String, L) passSelf() {...o.m(this)...}  
class XPatient L extends LPatient {...}  
class HPatient H extends XPatient {  
    //inherits passSelf()  
}
```

- ◆ Require: **H**-subclass of **L**-class overrides all inherited methods.
- ◆ Restrictive: Why override `getName`?
`(String, L) getName() {return this.name;}`

Example: Aliasing (2)

```
class LPatient L extends Object {  
    ...  
    (String, L) passSelf() {...o.m(this)...}  
class XPatient L extends LPatient {...}  
class HPatient H extends XPatient {  
    //inherits passSelf()  
}
```

- ◆ Require: **H**-subclass of **L**-class overrides all inherited methods.
- ◆ Restrictive: Why override `getName`?
`(String, L) getName() {return this.name;}`
- ◆ Use anonymous method(?) “this” not leaked...

Example: Control Flow (Conditional)

```
class XPatient L extends LPatient { //hiv, getHIV, setHIV}
    String    leakStatus() {
        var String    s; //level of s???
        if (this.hiv) {s := 'YES';} else {s := 'NO';}
        return s;
    }
}
```

Example: Control Flow (Conditional)

```
class XPatient L extends LPatient { //hiv, getHIV, setHIV }
(String, H) leakStatus() {
    var (String, H) s; //level of s???
    if (this.hiv) {s := 'YES';} else {s := 'NO';}
    return s;
}
```

If guard is **H**, only **H**-variables and **H**-fields may be modified.

Example: Control Flow (Dynamic Dispatch)

```
class XPatient L extends LPatient { //hiv, ...
```

```
class YN L extends Object {(bool, L)val()} {return true;}  
class Y L extends Object {(bool, L)val()} {return true;}  
class N L extends Object {(bool, L)val()} {return false;}}
```

Example: Control Flow (Dynamic Dispatch)

```
class XPatient L extends LPatient { //hiv, ...
    (YN, H) leak() {
        var (YN, H) o;
        if (this.hiv) {o := new Y();} else {o := new N();}
        return o;}}

class YN L extends Object {(bool, L)val()} {return true;}
class Y L extends Object {(bool, L)val()} {return true;}
class N L extends Object {(bool, L)val()} {return false;}}
```

Example: Control Flow (Dynamic Dispatch)

```
class XPatient L extends LPatient { //hiv, ...
  (YN, H) leak() {
    var (YN, H) o;
    if (this.hiv) {o := new Y();} else {o := new N();}
    return o;}}
xp.Leak() : (YN, H);

class YN L extends Object {(bool, L)val()} {return true;}
class Y L extends Object {(bool, L)val()} {return true;}
class N L extends Object {(bool, L)val()} {return false;}}
```

Example: Control Flow (Dynamic Dispatch)

```
class XPatient L extends LPatient { //hiv, ...
    (YN, H) leak() {
        var (YN, H) o;
        if (this.hiv) {o := new Y();} else {o := new N();}
        return o;}}
xp.Leak() : (YN, H); xp.Leak().val() : (bool, ???)
class YN L extends Object {(bool, L)val()} {return true;}
class Y L extends Object {(bool, L)val()} {return true;}
class N L extends Object {(bool, L)val()} {return false;}}
```

Example: Control Flow (Dynamic Dispatch)

```
class XPatient L extends LPatient { //hiv, ...
    (YN, H) leak() {...}
}

xp.Leak() : (YN, H)
xp.Leak().val() : (bool, H)
```

If level of receiver **H**, level of returned result from method call **H**.

Example: Dynamic Dispatch – Leaks via Heap

```
class YNh L extends Object {
    bool v;
    bool val() {return this.v;}
    unit setv(bool w) {this.v := w;}
    unit set() {this.setv(true);}
}

class Yh L extends YNh {unit set() {this.setv(true);}}
class Nh L extends YNh {unit set() {this.setv(false);}}
```

Example: Dynamic Dispatch – Leaks via Heap

```
class YNh L extends Object {
  bool v; //level of v???
  bool val() {return this.v;}
  unit setv(bool w) {this.v := w;}
  unit set() {this.setv(true);}
}

x := xp.Leak(); //x:(YNh, H)
x.set();

class Yh L extends YNh {unit set() {this.setv(true);}}
class Nh L extends YNh {unit set() {this.setv(false);}}
```

Example: Dynamic Dispatch – Leaks via Heap

```
class YNh L extends Object {
  bool v; //level of v???
  bool val() {return this.v;}
  unit setv(bool w) {this.v := w;}
  unit set() {this.setv(true);}
  y := new YNh() /*y:(YNh, L)*/ x := xp.leak(); //x:(YNh, H)
  x.set(); ...y.val()...
}

class Yh L extends YNh {unit set() {this.setv(true);}}
class Nh L extends YNh {unit set() {this.setv(false);}}
```

Example: Dynamic Dispatch – Leaks via Heap

```
class YN_h L extends Object {  
  (bool, H) v;  
  (bool, H) val() {return this.v;}  
  (unit, L) setv((bool, H) w) {this.v := w;}  
  (unit, L) set() {this.setv(true);}  
  
  x := xp.Leak(); x.set(); ...  
  y.val() :H ...
```

If level of receiver **H**, only **H**-fields may be modified in meth. call

Pointer Confinement

- ◆ L-object may be aliased by L-var, H-var
- ◆ L-class may have H-subclass
- ∴ Show L-Confinement:
 1. L-vars, L-fields do not contain H-pointers.
 2. Meaning of L-expression never H-pointer.

Pointer Confinement

- ◆ L-object may be aliased by L-var, H-var
- ◆ L-class may have H-subclass
- ∴ Show L-Confinement:
 1. L-vars, L-fields do not contain H-pointers.
 2. Meaning of L-expression never H-pointer.
 - ◆ In *conditionals/dyn. dispatch*, assignment may be confined to H-vars, H-fields.
- ∴ Show H-Confinement:

Input states, output states *indistinguishable* by L.

Formalisation

Types:

$\kappa ::= L \mid H$

$T ::= \text{unit} \mid \text{bool} \mid C$

$\tau ::= (T, \kappa)$ //security type

Formalisation

Types:

$\kappa ::= L \mid H$

$T ::= \text{unit} \mid \text{bool} \mid C$

$\tau ::= (T, \kappa) \text{ //security type}$

Typing Judgements:

$\Delta; C \vdash e : (T, \kappa) \text{ //}\Delta \text{ security type context}$

$\Delta; C \vdash S : (\text{com } \kappa_1, \kappa_2)$

Formalisation

Types:

$$\begin{aligned} \kappa & ::= L \mid H \\ T & ::= \text{unit} \mid \text{bool} \mid C \\ \tau & ::= (T, \kappa) \quad // \text{security type} \end{aligned}$$

Typing Judgements:

$$\begin{aligned} \Delta; C \vdash e & : (T, \kappa) \quad // \Delta \text{ security type context} \\ \Delta; C \vdash S & : (\text{com } \kappa_1, \kappa_2) \end{aligned}$$

“assign to vars $\geq \kappa_1$, update fields $\geq \kappa_2$ ”

Formalisation

Meanings of Typing Judgements:

$$\llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta h}, \quad \llbracket \Delta^\dagger; C \vdash S : \text{com} \rrbracket_{\mu\eta h}$$

$\mu \equiv$ Method Environment $\eta \equiv$ Stack $h \equiv$ Heap $(\eta, h) \equiv$ State

$$\eta \in \llbracket \Delta^\dagger \rrbracket$$

Formalisation

Meanings of Typing Judgements:

$$\llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta h}, \quad \llbracket \Delta^\dagger; C \vdash S : \text{com} \rrbracket_{\mu\eta h}$$

$\mu \equiv$ Method Environment $\eta \equiv$ Stack $h \equiv$ Heap $(\eta, h) \equiv$ State
 $\eta \in \llbracket \Delta^\dagger \rrbracket$

Related States $(\eta, h) \sim (\eta', h')$ are *indistinguishable* by L .

$\eta \sim \eta'$, iff $\forall x \in \text{dom } \Delta$, if $(T, L) = \Delta \ x$ then $\eta \ x = \eta' \ x$;

$h \sim h'$ iff same L -locations and those have equal L -fields.

Safe Expressions

Suppose:

- ◆ $\Delta; C \vdash e : (T, L)$
- ◆ $(\eta, h) \sim (\eta', h')$
- ◆ μ, η, η', h, h' are **L**-confined
- ◆ *safe* μ

◆ $\llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta h} \neq \perp \neq \llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta' h'}$

Then: $\llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta h} = \llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta' h'}$

Safe Expressions

Suppose:

- ◆ $\Delta; C \vdash e : (T, L)$
- ◆ $(\eta, h) \sim (\eta', h')$
- ◆ μ, η, η', h, h' are **L**-confined
- ◆ *safe* μ (i.e., method call in **L**-confined μ, η, η', h, h' , $(\eta, h) \sim (\eta', h')$, yields related heaps and (if non-**L**) returns equal results if return type of method is **L**)
- ◆ $\llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta h} \neq \perp \neq \llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta' h'}$

Then: $\llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta h} = \llbracket \Delta^\dagger; C \vdash e : T \rrbracket_{\mu\eta' h'}$

Safe Commands

Suppose:

- ◆ $\Delta; C \vdash S : (\text{com } k_1, k_2)$
- ◆ μ is **H**-confined
- ◆ ...same as for expressions ...
- ◆ $\llbracket \Delta^\dagger; C \vdash S : \text{com} \rrbracket_{\mu\eta} h \neq \perp \neq \llbracket \Delta^\dagger; C \vdash S : \text{com} \rrbracket_{\mu\eta} h'$.

Then **output states related**: $(\eta_0, h_0) \sim (\eta'_0, h'_0)$ where

$$(\eta_0, h_0) = \llbracket \Delta^\dagger; C \vdash S : \text{com} \rrbracket_{\mu\eta} h$$

$$(\eta'_0, h'_0) = \llbracket \Delta^\dagger; C \vdash S : \text{com} \rrbracket_{\mu\eta'} h'$$

Ongoing/Future Work

- ◆ Extension to full JavaCard
- ◆ Extension to full Java
 - ◆ **Threads**
 - ◆ **Generics**
 - ◆ ...
- ◆ Termination-insensitivity
- ◆ Inference of annotations (Pottier et. al.)
- ◆ Declassification (Halpern&O'Neill, Myers&Zdancewic)